BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots

Bogdan Nicolae, Franck Cappello

BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots

Bogdan Nicolae
INRIA Saclay, Île-de-France, France
bogdan.nicolae@inria.fr

Franck Cappello
INRIA Saclay, Île-de-France, France
University of Illinois at Urbana Champaign, USA
fci@lri.fr

ABSTRACT

Infrastructure-as-a-Service (IaaS) cloud computing is gaining significant interest in industry and academia as an alternative platform for running scientific applications. Given the dynamic nature of IaaS clouds and the long runtime and resource utilization of such applications, an efficient checkpoint-restart mechanism becomes paramount in this context. This paper proposes a solution to the aforementioned challenge that aims at minimizing the storage space and performance overhead of checkpoint-restart. We introduce an approach that leverages virtual machine (VM) disk-image multi-snapshotting and multi-deployment inside checkpoint-restart protocols running at guest level in order to efficiently capture and potentially roll back the complete state of the application, including file system modifications. Experiments on the G5K testbed show substantial improvement for MPI applications over existing approaches, both for the case when customized checkpointing is available at application level and the case when it needs to be handled at process level.

Categories and Subject Descriptors

D.3.4 [**Systems and Software**]: Distributed systems

General Terms

Design, Performance, Experimentation

Keywords

scientific computing, cloud computing, science clouds, IaaS, fault tolerance, checkpoint-restart, disk snapshots, MPI applications, capture application state

1. INTRODUCTION

In recent years, Infrastructure as a Service (IaaS) cloud computing [8] has emerged as a viable alternative to the acquisition and management of physical resources. With IaaS, users can lease storage and computation time from large datacenters. Leasing of computation time is accomplished by allowing users to deploy virtual machines (VMs) on the datacenter's resources. Since the user has complete control over the configuration of the VMs using on-demand deployments [6], IaaS leasing is equivalent to purchasing dedicated hardware but without the long-term commitment and cost.

Because of these advantages, cloud computing is gaining increasing attention for a wide range of scientific high performance computing (HPC) applications: climate modeling, bioinformatics, high-energy and nuclear physics, etc. Traditionally these applications run on powerful dedicated supercomputers, however recent evidence shows that there is an increasing improvement in the scalability and performance of cloud-based HPC systems [19]. Furthermore, unlike supercomputers, IaaS clouds enable scientists to fully customize the environment where the applications are running, as well as to easily share applications and input data globally, thus encouraging collaboration. These advantages, together with promising cost reductions, have instigated several initiatives to build science clouds, such as NASA's *Nebula* [3] and DoE's *Magellan* [26].

Since HPC applications require a lot of resources and clouds are mostly build out of commodity hardware [6], the number of components that can fail at any given moment in time is very high. Thus, an assumption about complete reliability is highly unrealistic: at such large scale, hardware component failure is the norm rather than the exception [33]. In this context, support for fault-tolerance becomes a critical issue.

A possible approach to deal with this issue is to use redundancy [11]. This approach is feasible if the benefits of increased resiliency outweigh the cost of consuming additional resources. However, for tightly coupled scientific applications, redundancy implies a replication of all processes that are part of the distributed application, as the failure of one process results in a global failure of all processes and leads to termination. Since clouds employ a pay-as-you-go model where the costs are directly proportional to the resource usage, such an approach is not feasible in our context.

Checkpoint-restart [15] is another widely used solution to provide fault tolerance for tightly coupled scientific applications. Processes achieve fault tolerance in this approach by saving recovery information periodically during failure-free execution. When a failure occurs, the previously saved recovery information can be used to restart the computation from an intermediate state, therefore reducing the amount of lost computation. This approach is highly appealing in the

SC'11, November 12–18, 2011, Seattle, Washington, USA.

context of clouds, as it does not consume more resources than strictly necessary. However, if failures happen often enough, the computation needs to repeatedly roll back to intermediate states, which increases resource usage and computation time. Therefore, it is crucial to design a scalable, high performance checkpoint-restart mechanism on clouds that is able to checkpoint the application frequently with minimal overhead, both with respect to performance overhead and storage space utilization.

This paper proposes *BlobCR*, a checkpoint-restart framework specifically optimized for tightly-coupled scientific applications that were written using a message passing system (in particular *MPI* [17]) and need to be ported to IaaS clouds. Our solution introduces a dedicated checkpoint repository that is able to efficiently take incremental snapshots of the whole disk attached to the virtual machine instances, thus offering support to use any checkpointing protocol that can save the state of processes into files, including application-level mechanisms, where the process state is managed by the application itself, and process-level mechanisms, where the process state is managed transparently at the level of the message passing library.

Our contributions can be summarized as follows:

- We present a series of design principles that facilitate checkpoint-restart on IaaS clouds and show how they can be applied in IaaS cloud architectures. Unlike conventional approaches, our proposal introduces support for an important feature: the ability to roll back I/O operations performed by the application. (Sections 3.1 and 3.2)
- We show how to materialize these design principles in practice by implementing checkpoint-restart framework based on a series of building blocks that rely on BlobSeer, a versioning storage service specifically designed for high throughput under concurrency [22, 23]. (Section 3.3)
- We evaluate our approach in a series of experiments, each conducted on hundreds of nodes provisioned on the Grid'5000 testbed, using both synthetic benchmarks and real-life applications. These experiments demonstrate significant improvement in performance and storage space utilization when compared to state-of-art. (Section 4)

2. INFRASTRUCTURE AND APPLICATION MODEL

Several properties of the infrastructure and the applications we target play a critical role in the design of an efficient checkpoint-restart mechanism. In this section we model both the infrastructure and the application, while insisting on such properties.

2.1 Cloud infrastructure

IaaS platforms are typically built on top of clusters made out of loosely-coupled commodity hardware [6]. Each machine is equipped with local disk storage in the order of several hundred GB, while interconnect is provided by mainstream networking technology, such as Ethernet. All machines have hardware virtualization support and run a hypervisor that is able to take advantage of it in order to efficiently host the VM instances of the users. We assume machines fail according to the *fail-stop* model, i.e. once a machine has failed, all hosted virtual machine instances and locally stored data are lost.

The local disk storage is typically used to hold the root file system of the instances, however each file system only persists during the life of its corresponding instance, as space is reclaimed after termination in order to host new instances. In order to provide persistent storage, a dedicated repository is separately deployed either as either a centralized [4] or as a distributed [7] storage service. This repository holds the VM images and application data of the users persistently and provides the means for users to manipulate them: upload, download, delete, and so forth.

2.2 Application model

We target tightly-coupled scientific applications that use a message-passing system to communicate. Hence, we assume that the application consists of a fixed number of processes that communicate through messages.

Such applications are typically designed to use a parallel file system for persistency (e.g., GPFS [30], PVFS [13]), which is used to read input data, dump output data and possibly save additional logging information or other intermediate data into files. In theory, the processes could also synchronize through the file system by sharing files. However, in practice this approach is avoided for scalability reasons, as it may lead to I/O bottlenecks. Most of the time, each process manipulates its own set of files independently of the other processes. Therefore, for the rest of this paper we assume that the processes do not need to share files except for the input data.

At a first glance, it may seem as if the cloud repository could play the role of a parallel file system and provide persistency for the application. However, repositories on clouds mostly offer a different access models for user data (e.g. key-value stores based on REST-ful access APIs [7], database management systems [20], etc.). Using a different access model requires significant changes to the application and is not always feasible (e.g. it is difficult to simulate a log file where data can be appended).

A much simpler solution is to rely on the local file system of the VM instance directly in order to share input files and save any output files and intermediate data. This approach does not require any changes to the application. Furthermore, it does not use any additional resources from the cloud, as the local file system of the VM instance is implicitly available to the user for storage. For the rest of this paper, we assume that such an approach is used to store the files of the application.

It is important to note that the local file system of the VM instance is not persistent and therefore the changes can be lost at any time, which implies the need to build an external persistency mechanism that is able to survive failures and can checkpoint the state of the local file systems for each VM instance. Using such an approach has an important advantage: the changes performed on the file system since the last checkpoint can be easily rolled back. This is not the case when using conventional storage solutions (i.e. the cloud repository or a parallel file system): for example, lines appended to a log file between the last checkpoint and the occurrence of a failure are difficult to detect and delete on restart.

2.3 Application state

Checkpoint-restart approaches achieve fault tolerance by periodically saving the global state of the application persistently to stable storage and restarting from an intermediate state in case of failures.

In the most general case, the state of the computation is defined at each moment in time by two main components: (1) the state of each of the VM instances; and (2) the state of the communication channels between them (opened sockets, in-transit network packets, virtual topology, etc.).

Thus, the general case implies saving both the state of all VM instances and the state of all active communication channels among them. While several methods have been established in the virtualization community to capture the state of a running VM (CPU registers, RAM, state of devices, etc.), the issue of capturing the global state of the communication channels is difficult and still an open problem [21].

In order to avoid this issue, the general case is usually simplified such that the application is reduced to the sum of states of the VM instances. This is consistent with many checkpoint-restart protocols [15], where in-transit network traffic is discarded under the assumption that a fault-tolerant communication protocol is used that is able to restore communication channels and resend lost information.

Even so, saving the whole state of the VM instances can explode to huge sizes and become unfeasible. For example, saving 2 GB of RAM for 1,000 VMs consumes 2 TB of space, which is unacceptable for a single one-point-in-time checkpoint. Therefore, there is a need to further reduce the state size for each VM instance.

In our context, it can be observed the state of each VM instance is defined by two components: (1) the state of the processes that perform the computation; and (2) the state of the file system of the guest operating system where the processes are running. Therefore, limiting the state to these two components has a high potential to reduce the overall state size. We focus our work in this direction.

3. OUR APPROACH

3.1 Design overview

Our proposal relies on the following key principles:

3.1.1 *Build a dedicated checkpoint repository using the local disks of compute nodes*

In many cloud deployments [6, 4, 5], the disks locally attached to the compute nodes are not exploited to their full potential. These disks have capacities of hundreds of GB yet the VM instances deployed on the compute nodes utilize just a fraction of it. Therefore, we propose to aggregate parts of the storage space from the compute nodes in a shared common pool that is managed in a distributed fashion. This pool is used to persistently store both the base VM images uploaded by the user and the checkpoints. For scalability reasons, they are stored in a striped fashion, i.e. split into small, equal-sized chunks that evenly distributed among the local disks of the checkpoint repository. Each chunk is replicated on multiple local disks in order to survive failures.

Using this scheme, read and write access performance under concurrency is greatly enhanced, as the global I/O workload is evenly distributed among the local disks. Furthermore, this scheme has a potential for high scalability, as a growing number of compute nodes automatically leads to a larger checkpoint repository, which is not the case if the checkpoints are saved directly on the cloud repository. Finally, it eases the pressure on the cloud repository, which can therefore provide improved performance and quality of service guarantees for the cloud applications that are specifically designed to make use of it.

3.1.2 *Use VM disk snapshots to checkpoint the application state*

In order to checkpoint both the local file system and application processes, we propose a two-stage procedure that is performed individually for each VM instance.

In the first stage, the application state is saved to the VM disk. This is done either explicitly, by relying on a custom checkpoint-restart mechanism at application level, or transparently, using an external process-level checkpointing protocol that is typically integrated in the message passing library, without requiring changes to be made to the application code. In the second stage, the VM instance is suspended and a snapshot of the virtual disk is saved persistently into the checkpoint repository, after which the VM instance is resumed.

The two-stage checkpoint procedure gives our proposal three key advantages. First, we avoid saving the state of the whole VM, which greatly reduces the state size: all memory used by the operating system, the information about the state of devices, etc. is discarded. Second, it provides an implicit roll-back mechanism for file system changes, as it is enough to simply restore the file system state from the disk-image snapshot. This avoids additional overhead present in many checkpointing techniques that log the interactions with the file system. Finally, it gives the user high flexibility to choose either an explicit or transparent technique.

It is important to note that the two stages occur in different environments: the first stage is performed inside the guest operating system of the VM instance, while the second stage is performed outside of the VM instance. As a consequence, there is a need to provide a synchronization mechanism that enables each VM instance to request a snapshot of its disk to the outside. This synchronization mechanism must be integrated into the checkpoint protocol. If checkpointing is explicitly handled at application level, then the application must be modified accordingly. Otherwise, the synchronization can be handled transparently at the level of the checkpoint protocol implemented inside the message passing system.

3.1.3 *Optimize VM disk snapshotting by means of shadowing and cloning*

Saving the full VM disk for each VM instance is not feasible in the context of checkpoint-restart. Since only small parts of the virtual disk are modified, this would mean massive unnecessary duplication of data, leading not only to an explosion of storage space utilization but also to an unacceptably high snapshotting time and network bandwidth utilization.

Several custom image file formats were proposed in order to avoid unnecessary duplication of data. Qcow2 [16] for example, one of the most popular choices implements incremental snapshotting by storing incremental differences as a separate file, while leaving the original file corresponding to the base disk image untouched and using it as a read-only

backing file. Using this approach, it is possible to create qcow2 images that are based on other qcow2 images in order to create a chain of "patches" that represent incremental differences. However, in order to do so the hypervisor needs to be restarted using a different underlying image. Therefore, such an approach cannot be applied in our context, as we need to take successive disk snapshots while the VM instance is still running, without restarting the hypervisor.

We propose a transparent solution to this problem that leverages two features used by versioning systems: *shadowing* and *cloning* [22].

Shadowing means to offer the illusion of creating a new standalone snapshot of the object for each update to it, but to physically store only the differences and manipulate metadata in such way that the illusion is upheld. This effectively means that from the user's point of view, if a small part of a large file needs to be updated, shadowing enables the user to see the effect of the update as a second file that is identical to the original except for the updated part.

Cloning means to duplicate an object in such way that it looks like a stand-alone copy that can evolve in a different direction from the original but physically shares all initial content with the original.

With this approach, snapshotting can be easily performed in the following fashion. The first time a snapshot is built, for each VM instance a new checkpoint image is cloned from the initial backing image. Subsequent local modifications are written as incremental differences to the checkpoint image and shadowed as a new snapshot. In this way all snapshots of all VM instances share unmodified content among one another and still appear to the outside as independent, fully fledged disk-images. This has an important advantage: differences are not stored as separate files and thus checkpoints are much easier to migrate. Furthermore, any additional overhead resulting from the need to assemble the checkpoints from multiple files during restart is avoided.

3.1.4 Optimize restart using lazy transfer and adaptive prefetching

Since our approach avoids saving the whole state of the VM instances, a restart implies that the instances are redeployed and rebooted using the disk snapshots of the last checkpoint, after which the state of the processes is restored from the files.

However, deploying a large number of instances concurrently can incur a significant overhead. Current techniques broadcast the disk images to the nodes before booting the VM instances [34], a process that can take tens of minutes to hours, not counting the time to boot the operating system itself. As VM instances typically access only a small fraction of the VM image throughout their run-time, fetching only the necessary parts on-demand can reduce this overhead considerably [24]. Therefore, we propose the use of a "lazy" transfer scheme that fetches only the hot content of the disk image (i.e. the checkpoint files and any other files directly accessed at runtime by the guest operating system and the application).

Furthermore, since the disk snapshots store only incremental differences, large parts of the images are shared and potentially need to be accessed concurrently by the hypervisor during the boot process. In order to limit the negative impact of this issue, we exploit small delays between the times when the VM instances access the same chunk from the checkpoint repository (due to jitter in execution time) in order to prefetch the chunk for the slower instances based on the experience of the faster ones [25].

3.2 Architecture

The simplified architecture of an IaaS cloud that integrates our approach is depicted in Figure 1. The typical elements found in the cloud are illustrated with a light background, while the elements that are part of our proposal are highlighted by a darker background. Except for the guest environment (VM) that is under the control of the user, all other highlighted building blocks must be adopted by the cloud provider.

A *checkpoint repository* that survives failures and supports cloning and shadowing is deployed on the compute nodes. The checkpoint repository aggregates part of the storage space provided by the local disks of the compute nodes and is responsible to persistently store both the base and the checkpoint disk images.

The *cloud client* has direct access to the checkpoint repository and is allowed to upload and download the disk images. Typically the user downloads and uploads base disk images only, however, thanks to shadowing and cloning, our approach enables the user to see and download checkpoint images as standalone entities as well. This feature that can become useful in a scenario where the checkpoints need to be inspected and even manually modified. Moreover, the cloud client interacts with the *cloud middleware* (the frontend of the user to the cloud) through a control API that enables multi-deployments of a large number of VM instances starting from the same base disk image.

Each compute node runs a *hypervisor* that is responsible to launch and execute the VM instances. The VM instances run in a modified guest environment that implements an extended checkpoint-restart protocol, which is able to communicate with the hosting compute nodes and ask each of them to freeze the state of the VM, take a checkpoint of the virtual disk and then continue VM execution. This is done through the *checkpointing proxy*, a special service that runs on the compute nodes and accepts checkpoint requests. Both for security and scalability reasons, the checkpointing proxy is not globally accessible: it accepts checkpoint requests only from the VM instances that are hosted on the same compute node.

All reads and writes issued by the hypervisor are trapped by the *mirroring module*, responsible to fetch the hot contents of the base disk image remotely from the repository and cache it locally. Local modifications to the base disk image triggered by writes are stored on the local disk as incremental differences. Whenever a checkpoint request is issued for the first time, the checkpointing proxy asks the mirroring module to create a checkpoint image that is derived from the base image (CLONE). This initial checkpoint image shares all contents with the base image. Then, the local modifications are committed to the checkpoint image as an incremental snapshot (COMMIT). Any subsequent checkpoint request will commit the local modifications recorded since the last checkpoint request as a new incremental snapshot into the same checkpoint image.

A mapping between each successful checkpoint request and the resulting incremental snapshot together with its corresponding checkpoint image is maintained by the cloud middleware. In case of a failure or when the whole applica-

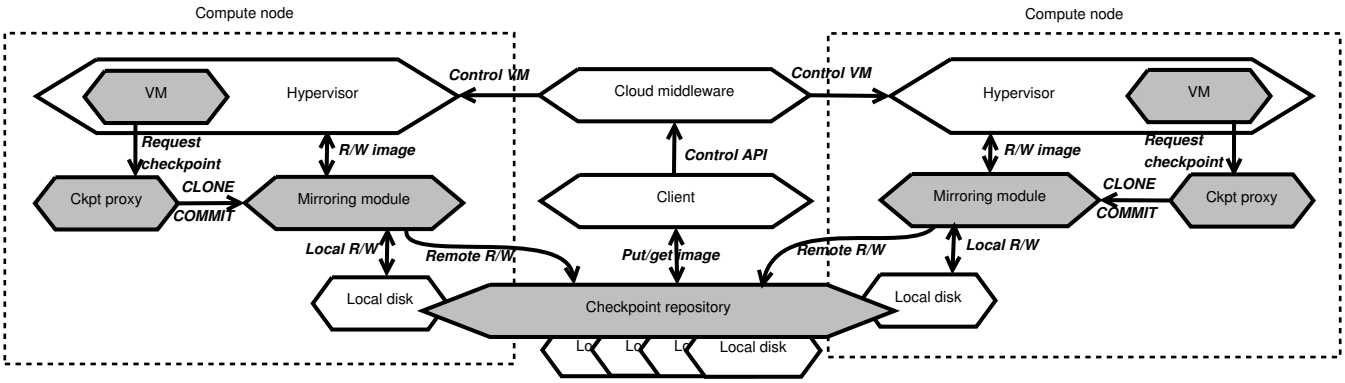


Figure 1: Our approach (dark background) integrated in an IaaS cloud.

tion needs to be terminated and resumed at a later point, all VM instances are re-deployed using a recent snapshot from their corresponding checkpoint image as the underlying virtual disk. It is the responsibility of the checkpoint-restart protocol implementation to pick a set of snapshots for the VM instances such that the application can roll back to a globally consistent state.

3.3 Implementation

In Section 3.2 we illustrated how to apply our approach in the cloud by orchestrating several building blocks: a distributed checkpoint repository, a mirroring module, a checkpointing proxy and a modified checkpoint-restart protocol running inside the VM instances that decides when to request disk-image snapshots. In this section we show how to efficiently implement these building blocks in such a way that they achieve the design principles introduced in Section 3.1 on the one hand and are easy to integrate in the cloud on the other hand.

We have implemented the distributed checkpoint repository on top of *BlobSeer* [22, 23]. This choice was motivated by several factors. First, BlobSeer enables *scalable aggregation of storage space* from the participating nodes with minimal overhead in order to store *BLOBs* (Binary Large OBjects). Data striping and replication is performed transparently on BLOBs, which enables direct mapping between BLOBs and disk-images, therefore eliminating the need to explicitly manage chunks. Second, BlobSeer offers out-of-the-box support for shadowing, which significantly simplifies the implementation of the COMMIT primitive. Third, the service was optimized to sustain a high throughput even under heavy access concurrency, which is especially useful in our context, as it enables efficient parallel access to the chunks when disk image snapshots need to be read or written.

The *mirroring module* was implemented on top of *FUSE* (File System in UserspacE) [2], and relies on our previous work presented in [24]. It exposes each checkpoint image as a directory and the associated snapshots as files in that directory, accessible from the outside using the regular POSIX access interface. Internally, the module keeps track of the content that is available locally, as well as the local modifications. It translates each read and write request originating from the hypervisor, respectively, into local/remote reads (depending on whether the content is available locally or not) and into local writes.

This approach presents several advantages. First, it enables exposing the VM image to the hypervisors as a regular raw file that is accessible through the standard POSIX access interface. Second, it hides away both the copy-on-write management of local modifications to the disk-image as well as the on-demand mirroring. This is achieved by exposing a raw image file to the hypervisor, which ensures maximum compatibility with most hypervisors. Third, it transparently handles disk-image snapshotting, which avoids the need to stop the hypervisor and switch to a different image whenever a snapshot is needed. In order to make this possible, the CLONE and COMMIT primitives are implemented as *ioctls* and can be accessed from outside the hypervisor.

The *checkpointing proxy* was implemented as a service that listens on a specified port for incoming TCP/IP connections originating from VM instances that resides on the same compute node where the proxy is deployed. Whenever a connection is initiated and a checkpoint is requested, the proxy authenticates the VM instance and, if successful, proceeds to: (1) suspend the VM instance; (2) clone the base image if necessary using the appropriate ioctl; (3) commit the local changes as a new snapshot inside the checkpoint image (again using the appropriate ioctl); and finally (4) resume the VM instance. Regardless whether the checkpoint was successful or not, the proxy resumes the VM instance and notifies it of the result.

If application-level checkpointing is desired, the checkpointing proxy can be directly contacted from within the application code. For maximum compatibility, the communication protocol used by the proxy is a simple REST-ful access interface. In order to implement process-level checkpointing in a transparent fashion, we provide a modified MPI [17] library implementation based on *mpich2* that must be installed in the guest operating system. The mechanism implemented in mpich2 is a coordinated checkpointing protocol that is executed in three steps. First, the communication channels are drained such that no in-transit messages are lost. This is performed by sending a special marker message to each MPI process, instructing it not to send any message from that point on until the checkpoint has completed. Next, *blcr* [14] is used on each VM instance to dump the checkpoint of the MPI processes into files. Finally, once all MPI processes have been successfully checkpointed, the MPI library resumes application execution.

We extended the coordinated checkpointing protocol with two additional steps: immediately after the process state

was dumped by blcr as a file, the sync system call is invoked in order to flush all uncommitted changes to the virtual disk. This is necessary in order to avoid any potential file system corruption due to caching. After this step completed, a checkpoint request is sent to the checkpoint proxy. As soon as confirmation is received, control is returned to the original mpich2 implementation that proceeds to resume application execution.

4. EVALUATION

This section evaluates the benefits of our proposal both in synthetic settings and for real-life applications.

4.1 Experimental setup

The experiments were performed on Grid'5000 [10], an experimental testbed for distributed computing that federates nine sites in France. We used 120 nodes of the graphene cluster from the Nancy site, each of which is equipped with a quadcore Intel Xeon X3440 x86_64 CPU with hardware support for virtualization, local disk storage of 278 GB (access speed $\simeq$55 MB/s using SATA II ahci driver) and 16 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of $\simeq$0.1 ms).

The hypervisor running on all compute nodes is KVM 0.14.0, while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 2 GB raw disk image file based on the same Debian Sid distribution was used as the guest operating system. Inside this guest OS, we installed a modified mpich2 library (based on the 1.3.x development branch) that integrates our approach.

4.2 Methodology

We use three settings for our evaluation:

4.2.1 *Application-level checkpointing using disk snapshots.*

In this setting, the application itself is responsible to store and restore the state of each process through files that are periodically saved in the file system of VM instances. A global checkpoint consists in taking a snapshot of all the virtual disks that hold the file systems of the VM instances. In order to restart from such a global checkpoint, the VM instances need to be re-deployed in such way that each instance is using one of the disk snapshots that are part of the global checkpoint.

To take a disk snapshot, two alternative approaches are used:

BlobCR.

This is our approach: BlobSeer is used as the distributed repository, along with the FUSE-based implementation of the mirroring module, as described in Section 3.3. We deploy a version manager and a provider manager, each on a dedicated node, along with 20 metadata providers, again each on a dedicated node. The rest of 120 nodes are used as compute nodes. A data provider, a mirroring module and a checkpointing proxy is then launched on each compute node. The initial raw disk image that holds the guest OS is stored into the BlobSeer deployment in a striped fashion. The stripe size was fixed at 256KB, as we found this to maximize the trade-off resulting from the need to choose a small stripe size in order to reduce access contention vs. the need to keep the stripe size large enough in order to avoid excessive fragmentation overhead.

Throughout the rest of this section, we refer to this approach as *BlobCR-app*.

Qcow2 disk snapshots over PVFS.

We compare our approach to the case when the virtual disk are based on qcow2 [16] disk snapshots and are persistently stored in a parallel file system. For the purpose of this work, we have chosen PVFS [13] as the parallel file system. This choice was mainly motivated by the fact that PVFS was specifically designed for high performance access patterns that do not exhibit conflicting concurrent writes to the same file - a scenario that applies in our context, since the VM instances do not modify the same image. PVFS is deployed on all nodes, out of which 120 are reserved as compute nodes. The initial raw disk image that holds the guest OS is stored in PVFS using a stripe size of 256KB, the same size as in our approach and is accessible on all compute nodes through a local mount point. Using the raw image as a backing file, a qcow2 disk snapshot is created on the local disk of the compute node for each VM instance, using the qemu−img tool. The qcow2 image is responsible to store the local modifications of the VM instance to the virtual disk that holds the guest file system. In order to take a disk snapshot at a particular moment in time, the checkpointing proxy simply copies the locally stored qcow2 image to PVFS as a new file.

We refer to this approach as *qcow2-disk-app*.

4.2.2 *Process-level checkpointing using disk snapshots.*

In this setting, checkpointing is orchestrated transparently by our MPI library implementation, as described in Section 3.3, i.e. the states of the processes are dumped to files using blcr.

The same two approaches as described above are used to take the disk snapshots. We refer to these two approaches as *BlobCR-blcr* and *qcow2-disk-blcr* respectively.

4.2.3 *VM checkpointing using full snapshots.*

This setting is similar to the previous setting except that the state of the processes is not dumped into files. Rather, for each instance a snapshot of the whole VM is taken. This snapshot includes not only the virtual disk, but all other devices of the VM as well: CPU registers, memory, etc. Such an approach in different in that the instance needs not be rebooted when a restart is required, but can be directly resumed from the complete VM snapshot.

In order to take full VM snapshots and store them persistently, we use *qcow2* images that are saved to PVFS. This setup is very similar to the case when qcow2 disk snapshots are used: we deploy PVFS under the same conditions and create a local qcow2 image for each VM instance, using the initial raw image (shared through PVFS) as a backing file. However, in this case the whole state of the VM is dumped to the qcow2 image using the savevm QEMU monitor command.

We refer to this approach as *qcow2-full*.

4.3 Synthetic benchmarks

The first series of experiments evaluates the scalability of our proposal in controlled synthetic settings.

To this end, we implemented a simple benchmarking application that consists of a configurable number of processes, each of which runs in a dedicated VM instance. Each process independently allocates a fixed amount of memory as a data buffer and fills it with random data. In order to take a global checkpoint at application level, the processes synchronize to start at the same time and then independently dump the data buffer into a file, after which they ask the checkpointing proxy to snapshot the disk. On restart, each process reads the contents of the previously saved file into the data buffer.

We study both checkpoint and restart, together with the storage space utilization for all five approaches: *BlobCR-app, BlobCR-blcr, qcow2-app, qcow2-blcr and qcow2-full*. The data buffer was fixed at two sizes: 50 MB and 200 MB.

4.3.1 Increasing number of processes

This experiment consists in concurrently deploying an increasing number of VM instances, each on a dedicated compute node. Once the instances have fully booted, the benchmarking application is launched and a global checkpoint is issued using one of the five approaches. We record the *completion time* to save the global checkpoint to persistent storage (i.e. the time elapsed between the moment when the checkpoint request was issued and the moment when all snapshots were successfully taken and persistently saved).

After this operation successfully completed, we simulate a restart by killing all instances and re-deploying them using the previously saved snapshots as the underlying images. To make sure that no caching effects interfere with the experiment, each instance is re-deployed on a different compute node that the one where it originally ran. Except for the *qcow2-full* approach, the instances reboot the guest operating system and then restore the state of the application from the previously saved files. Again, we record the completion time for the whole process, i.e. the time elapsed between the moment when the re-deployment begins and the moment when the state of all processes was successfully restored.

The completion time to checkpoint an increasing number of processes for a data buffer of 50 MB is represented in Figure 2(a). As can be observed, *qcow2-full* performs the worst out of the five approaches. This happens for two reasons: (1) saving the state of the VM instance is time-consuming; (2) the size of the resulting full snapshot is much larger than a disk-snapshot and therefore needs more time to transfer. The performance of all other four approaches is much better. A steady increase in completion time is noticeable when increasing the number of processes, which is caused by the increased write pressure under concurrency. *BlobCR-app* and *qcow2-disk-app* have very close performance levels, unlike the case of process-level checkpointing, where *BlobCR-blcr* ourperforms *qcow2-disk-blcr* under concurrency by almost 40%.

When increasing the data buffer from 50 MB to 200 MB (Figure 2(b)), our approach shows much better scalability: for 120 processes, *BlobCR-app* is 60% faster than *qcow2-disk-app*, while *BlobCR-blcr* reaches a two-fold speedup compared to *qcow2-disk-blcr*. These results are a consequence of the fact that BlobSeer tolerates higher write pressure under concurrency than PVFS, which becomes more visible with increasing size of the disk-snapshot. In the case of *qcow2-full*, the effect of higher write pressure is augmented even further, which enables our approach to outperform it by a factor of more than six.

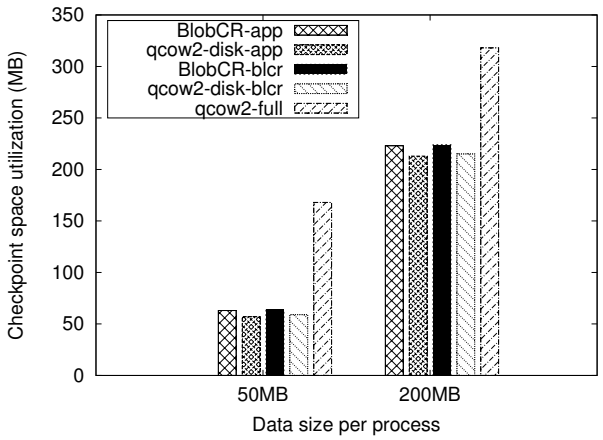
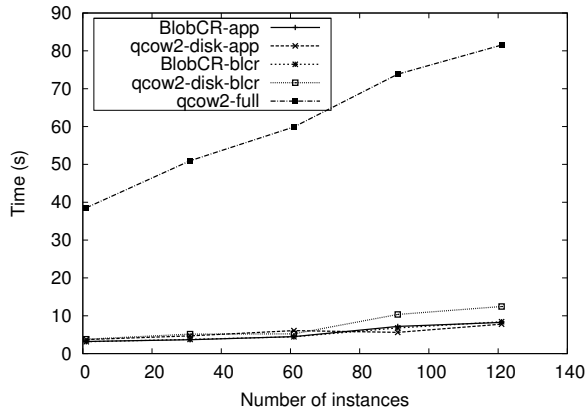


Figure 4: Snapshot size for a data buffer of 50 MB and 200 MB

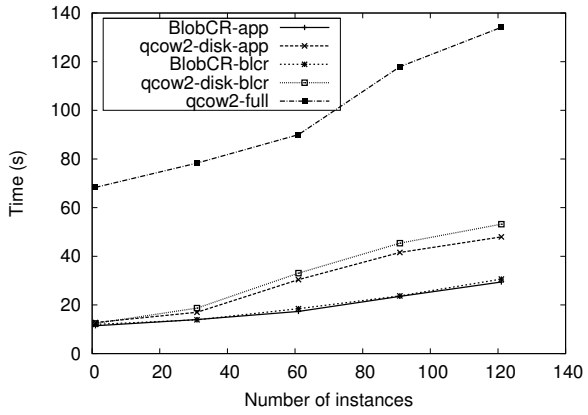
A similar trend can be observed for the completion time to restart the processes (Figure 3). Again, *qcow2-full* has the worst performance of all four approaches, despite avoiding the need to reboot the VM instance. This poor performance can be traced back to the fact that the full snapshot is much larger than a corresponding disk-snapshot and thus takes longer to read from PVFS, which ultimately cancels the benefit of avoiding a reboot. Application-level restart and process-level restart have very close performance levels, both for 50 MB (Figure 3(a)) and 200 MB (Figure 3(b)). Thanks to a faster reboot time, *BlobCR* is by more than 25% faster than *qow2-disk* for a data buffer of 50 MB. When increasing the data buffer from 50 MB to 200 MB, both the faster reboot time and the better sustained read throughput under concurrency enable our approach to remain highly scalable for increasing the number of processes, which ultimately leads to a speedup of 2x compared to *qow2-disk*.

Figure 4 depicts the size of the snapshot per VM instance when the process running inside it allocates a data buffer of 50 MB and 200 MB respectively. As expected, the minimal size is obtained when using application-level checkpointing. In addition to the file into which the process saved its state, the disk-snapshot holds also some minor updates to the file system that were performed by the guest operating system (i.e. configuration files generated at boot time, daemons writing to log files, etc.). These minor updates add up to 13 MB for *BlobCR-app* and 7 MB for *qcow2-disk-app*. Our approach has a slightly higher overhead, because differences are maintained at block level granularity and therefore cannot be smaller than 256 KB, whereas qcow2 can maintain arbitrarily small differences. Nevertheless, considering the case of 200 MB, the price to pay in terms of storage space remains constant and is less than 5% over qcow2, yet it brings a more than double performance speedup that shows a clear tendency to grow even larger for an increasing number of processes.

In the case of process-level checkpointing using blcr, the disk-snapshots see only a negligible increase in size (less than 2 MB) when compared to application-level checkpointing. This is the case for both *BlobCR-blcr* and *qcow2-disk-*

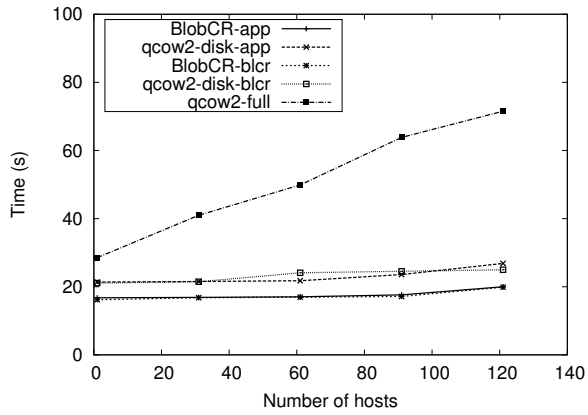


(a) Data buffer of 50 MB

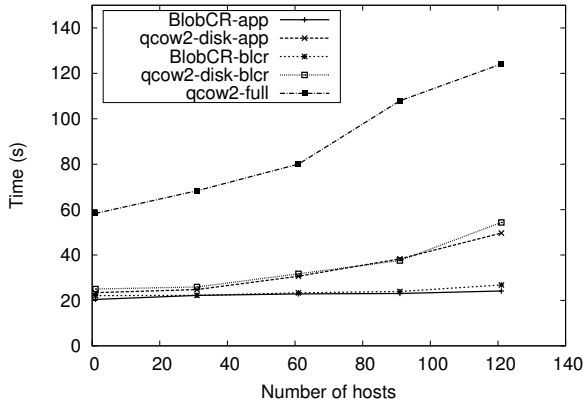


(b) Data buffer of 200 MB

Figure 2: Completion time to checkpoint an increasing number of processes



(a) Data buffer of 50 MB



(b) Data buffer of 200 MB

Figure 3: Completion time to restart an increasing number of processes

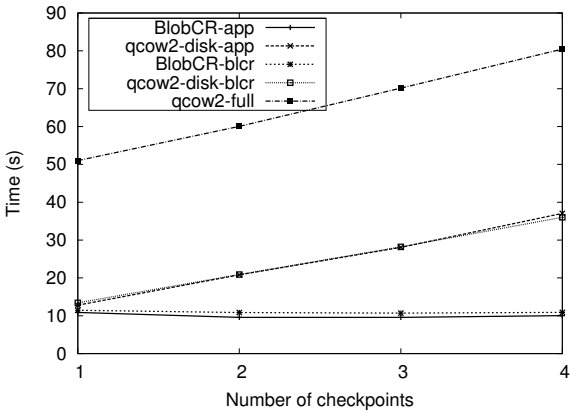
blcr and it leads to an important conclusion: when the state of the process comprises most of its allocated memory, then application-level checkpointing brings little benefit compared to process-level checkpointing.

Finally, the size of full VM snapshots is much larger than the size of disk-snapshots. Results show an overhead of 118MB, both for a data buffer of 50MB and 200MB respectively. This overhead accounts for the memory used by all other processes launched by the guest operating system, state of devices, operating system caches, etc. In our experiment, checkpointing was performed immediately after the benchmarking application was successfully initialized. Therefore, in a real life scenario where the application runs for long periods of time, even higher overheads are to be expected.

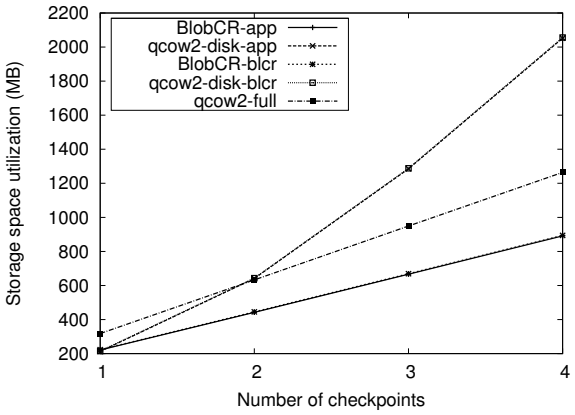
4.3.2 Successive checkpoints

Our next experiment evaluates the performance and storage space utilization for all five approaches when taking successive checkpoints of the same deployment. To this end, we deploy the benchmarking application and execute the following two steps for four times: (1) fill the data buffer with random data and then (2) request a global checkpoint. To limit the number of factors that can influence the results, in particular the impact of I/O pressure under concurrency (which was shown in the previous section to give our approach a large advantage), we experiment with a single VM instance, into which a single benchmarking process is launched. The size of the data buffer is fixed at 200 MB.

Results are shown in Figure 5. As can be observed, our approach has perfect scalability with respect to completion time (Figure 5(a)), thanks to the fact that only incremental differences are stored between snapshots. This applies for both *BlobCR-app* and *BlobCR-blcr*. In the case of *qcow2-disk-app* and *qcow2-disk-blcr*, a linear growth is clearly visible. This is due to the fact that the local qcow2 image grows larger as the application is running and thus takes longer to transfer to PVFS. It can be traced back to the lack of transparent snapshotting support, as explained in Section 3.1.3,



(a) Completion time



(b) Total storage space utilization

Figure 5: Four successive checkpoints of the same VM instance for a data buffer of 200 MB

and is responsible for the linear growth of completion time in the case of *qcow2-full* too.

The total storage space utilization for all five approaches is depicted in Figure 5(b). As expected, our approach grows linearly. The same applies to *qcow2-full*. This is due to the fact that a read-only incremental VM snapshot is created directly inside the original qcow2 image, which continues to act the the underlying image when the VM instance is resumed. Since an unlimited number of read-only snapshots can be saved inside the same qcow2 image, it is sufficient to persistently store only the latest version of the qcow2 image to PVFS. However, qcow2 does not support the same functionality for disk snapshots, which means that consecutive disk snapshots need to be stored as separate files and accumulate duplicate data, ultimately leading to an exponential growth in storage space.

4.4 Real life application case study: CM1

Our next series of experiments illustrates the behavior of our proposal in real life. For this purpose we have chosen *CM1*, a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. This application is used to study small-scale processes that occur in the atmosphere of the Earth, such as hurricanes.

CM1 is representative of a large class of scientific applications that model a phenomenon in time which can be described by a spatial domain that holds the value of fixed parameters in each point (temperature, pressure, etc.). Starting from such an initial spatial domain, the application calculates the evolution of the values of the parameters in each point according to a set of governing equations that involves the previous values of the parameters in that point and eventually its neighborhood. The problem is solved iteratively in a distributed fashion by splitting the spatial domain into subdomains, each of which is managed by a dedicated MPI process. At each iteration, the MPI processes calculate the values for all points of their subdomain, and then exchange the values at the border of their subdomains with each other.

CM1 is able to take application-level checkpoints by synchronizing the MPI processes to dump the contents of the subdomains into files. Each MPI process independently writes its own checkpoint file. Furthermore, at each fixed number of iterations, all MPI processes write intermediate summary information about the subdomains, again into independent files. For the purpose of this work, we have chosen a 3D hurricane that is a version of the Bryan and Rotunno simulations [12]. We study the weak scalability of our approach by solving the same problem using a different precision, in such way that the size of the subdomain solved by each process remains constant at 50x50.

The experiment consists in deploying an increasing number of quad-core VM instances, each of which hosts 4 MPI processes, one per core. We take a global checkpoint after 10 minutes of execution time and record the completion time and storage space utilization.

This experiment is performed both for application-level checkpointing as implemented by CM1 (*BlobCR-app* and *qcow2-disk-app*) as well as process-level checkpointing using blcr (*BlobCR-blcr* and *qcow2-disk-blcr*). We found the size of the full snapshot to grow to unacceptably large sizes, which puts *qcow2-full* at a disproportionately large disadvantage and thus was omitted.

The per disk-snapshot size is depicted in Table 1. Our approach has a slightly higher overhead, as explained in Section 4.3.1. Unlike the case of synthetic benchmarks, in this case process-level checkpointing has a much higher overhead than application-level checkpointing. This effect happens because blcr indiscriminately dumps all memory allocated by the process, whereas application-level checkpointing is more informed and selects only the useful information to be saved.

The small overhead in storage space of our approach makes up for better performance and scalability with respect to completion time, as shown in Figure 6. All four approaches exhibit an increasing tendency for a growing number of processes. This tendency is slightly higher than for our benchmarking application, because the processes take longer to synchronize (i.e. wait for communication channels to be flushed, etc.). The difference between BlobCR and qcow2-disk grows higher as more processes are added (both for *app* and *blcr*), which demonstrates better scalability for our ap-

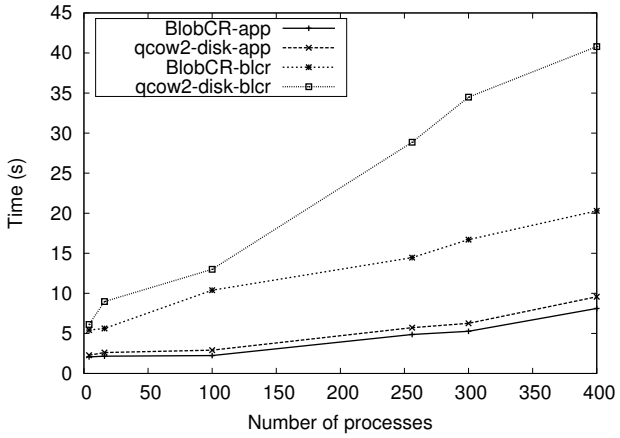


Figure 6: CM1 checkpoint performance for an increasing number of processes

Table 1: CM1 per disk snapshot size

Approach	Size
BlobCR-app	52 MB
qcow2-disk-app	45 MB
BlobCR-blcr	127 MB
qcow2-disk-blcr	120 MB

proach. At 400 processes, *BlobCR-app* outperforms *qcow2-disk-app* by more than 10%, while *BlobCR-blcr* outperforms *qcow2-disk-blcr* by more than a factor of 2.

5. RELATED WORK

The idea of using a dedicated checkpoint repository in order to optimize for the access patterns that are present in the context of checkpoint-restart has been exploited before. In [9], the authors propose PLFS, a virtual parallel log structured file system specifically designed for checkpoint storage. In essence it remaps an application's preferred data layout into one which is optimized for the underlying parallel file system. Unlike our approach, PLFS is a layer of indirection and thus is is heavily dependent on the performance characteristics of the parallel file system.

Attempts to virtualize the environment were undertaken by DejaVu [29], a transparent user-level framework for MPI applications that virtualizes the OS interface, making it transparent to both applications and communication middleware. It is based on a potentially expensive, on-line logging protocol which relaxes the requirements of a distributed snapshot and implements a reliable communication protocol.

Closer to our approach are checkpoint-restart proposals based on full VM snapshots [32, 31]. Unlike our approach, they focus on complete transparency using the Xen hypervisor and do not support incremental snapshotting. To our best knowledge, we are the first to propose a checkpoint-restart framework for HPC applications based on incremental disk snapshots, which has the potential to drastically reduce the storage space utilization at the cost of minimal intervention inside the guest environment.

Many hypervisors provide native copy-on-write support using custom VM image file formats, such as *qcow2* [16] and *Mirage* [27]. This enables base images to be used as read-only templates for multiple VM disk snapshots that store per-instance modifications. However, unlike our approach, support for transparent incremental snapshotting (i.e. without switching to another image) is currently not available. Furthermore, lots of files representing incremental differences need to be generated and shared through a parallel file system, which raises manageability and performance issues at large scales.

Several other approaches have been proposed in order to snapshot virtual disks. Lithium [18] is one such approach. It supports fork-consistent, instant volume creation with lazy space allocation, instant creation of writable snapshots, and tunable replication. While this can prove a valuable building block that offers a viable alternative to cloning and shadowing, it is based on log-structuring [28], which can potentially incur a high read overhead the more incremental snapshots are taken.

Amazon EBS [1] provides block level storage volumes that can be attached to Amazon EC2 [6] instances. Such volumes outlive the VM instances that mount and use them, which makes them a potential target to store the process state and all other intermediate files. Snapshotting is supported, however it is implemented over Amazon S3 [7], a key-value store not specifically optimized for this purpose.

6. CONCLUSIONS

High-performance and scalability of checkpoint-restart are two crucial challenges that need to be addressed on IaaS clouds in order to bring fault tolerance for HPC applications that are deployed on such platforms. This paper has proposed *BlobCR*, a checkpoint restart framework specifically written to address these challenges. Unlike conventional approaches, our proposal introduces support for an important feature: the ability to roll back I/O operations performed by the application.

We demonstrated the benefits of our approach through experiments on hundreds of nodes using synthetic benchmarks as well as real-life applications. *BlobCR* brings a checkpointing time speedup of up to 8x compared to full VM snapshotting based on *qcow2* over PVFS, as well as a speedup of more than 2x when compared to disk-snapshotting based on qcow2 over PVFS. A similar trend is observable for restart times: a speedup of up to 6x is observed vs. full VM snapshotting, and a speedup of up to 2x is observed vs. disk-snapshotting. On top of these benefits, an additional advantage of our approach is the potential to save large amounts of storage space thanks to our transparent incremental snapshotting support.

Based on our experiments, we conclude that checkpointing the whole state of VM instances using full VM snapshots is expensive, both in terms of storage space and performance. Checkpoints using disk-only snapshots are much smaller and faster, even if a reboot of the guest operating system is needed on restart.

We plan to extend our approach in future work with additional features that can potentially bring further benefits at large scale. In particular, we plan to explore how transparent garbage collection would reclaim the space used by disk-snapshots that are obsoleted by newer checkpoints.

Acknowledgments

This work was supported in part by the Agence Nationale de la Recherche (ANR) under Contract ANR-10-01-SEGI and

the Joint Laboratory for Petascale Computing, an INRIA-UIUC initiative. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

7. REFERENCES

[1] Amazon Elastic Block Storage (EBS). http://aws.amazon.com/ebs/.

[2] File System in UserspacE (FUSE). http://fuse.sourceforge.net.

[3] Nasa nebula. http://nebula.nasa.gov.

[4] Nimbus. http://www.nimbusproject.org/.

[5] Opennebula. http://www.opennebula.org/.

[6] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[7] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[8] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[9] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the 22nd Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Portland, USA, 2009.

[10] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, 20:481–494, November 2006.

[11] R. Brightwell, K. Ferreira, and R. Riesen. Transparent redundant computing with mpi. In *EuroMPI'10: Proceedings of the 17th European MPI user's group meeting conference on recent advances in the message passing interface*, pages 208–218, Stuttgart, Germany, 2010.

[12] G. H. Bryan and R. Rotunno. The maximum intensity of tropical cyclones in axisymmetric numerical model simulations. *Journal of the American Meteorological Society*, 137:1770–1789, 2009.

[13] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, USA, 2000.

[14] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Future Technologies Group, 2002.

[15] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[16] M. Gagné. Cooking with Linux—still searching for the ultimate Linux distro? *Linux J.*, 2007(161):9, 2007.

[17] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1999.

[18] J. G. Hansen and E. Jul. Scalable virtual machine storage using local disks. *SIGOPS Oper. Syst. Rev.*, 44:71–79, December 2010.

[19] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. Case study for running hpc applications in public clouds. In *HPDC '10: Proceedings of the 19th International Symposium on High Performance Parallel and Distributed Computing*, pages 395–401, Chicago, USA, 2010.

[20] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[21] X. Liu, J. Huai, Q. Li, and T. Wo. Network state consistency of virtual machine in live migration. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 727–728, Sierre, Switzerland, 2010.

[22] B. Nicolae. *BlobSeer: Towards Efficient Data Storage Management for Large-Scale, Distributed Systems*. PhD thesis, University of Rennes 1, November 2010.

[23] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, 71:169–184, February 2011.

[24] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu. Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In *HPDC '11: The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 147–158, San Jose, USA, 2011.

[25] B. Nicolae, F. Cappello, and G. Antoniu. Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In *Euro-Par '11: Proceedings of the 17th International Euro-Par Conference on Parallel Processing*, Bordeaux, France, 2011.

[26] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu. Magellan: experiences from a science cloud. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 49–58, San Jose, USA, 2011.

[27] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *VEE '08: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 111–120, Seattle, USA, 2008.

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[29] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. Dejavu: transparent user-level checkpointing, migration and recovery for distributed systems. In *SC '06: Proceedings of the 19th Conference on High Performance Computing Networking, Storage and Analysis*, Tampa, USA, 2006.

[30] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, USA, 2002.

[31] G. Vallée, T. Naughton, H. Ong, and S. Scott. Checkpoint/restart of virtual machines based on Xen. In *HAPCW '06: Proceedings of the High Availability and Performance Workshop*, Santa Fe, USA, 2006.

[32] O. Villa, S. Krishnamoorthy, J. Nieplocha, and D. M. Brown, Jr. Scalable transparent checkpoint-restart of global address space applications on virtual machines over Infiniband. In *CF '09: Proceedings of the 6th ACM Conference on Computing Frontiers*, pages 197–206, Ischia, Italy, 2009.

[33] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204, Indianapolis, USA, 2010.

[34] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath. Image distribution mechanisms in large scale cloud providers. In *CloudCom '10: Proceedings 2nd IEEE International Conference on Cloud Computing Technology and Science*, Indianapolis, USA, 2010.